Rapport pour le projet n°1 de COMPLEX

ROUSSEAU Sylvain et WANG Jinxin

Octobre 2014

Introduction

Dans le cadre du projet, nous nous intéressons aux puces RFID. Les puces RFID sont des dispositifs de petites tailles pouvant stocker une quantité limitée de données. La mémoire des puces étant limitée, il faut recourir à une sélection des données que l'on va enregistrer. Pour ce faire on attribue à chaque donnée une utilité (à savoir une valeur dans la modélisation de notre problème). Pour chaque donnée, on a également accès à sa taille (le poids dans la modélisation de notre problème). On revient donc finalement au « problème du sac à dos ». Ce problème est NP-difficile. On s'intéressera à deux approches pour sa résolution : la programmation dynamique et des méthodes arborescentes. Dans les deux cas nous essaierons d'implémenter un algorithme exact et un problème approximant la solution. L'implémentation des algorithmes dont il sera question dans le rapport a été faite en Python.

Première partie

Programmation dynamique (algorithme de résolution pseudo-polynomial)

La programmation dynamique est un concept visant à s'abstraire d'une des règles de notre problème pour résoudre un sous-problème qui sera plus « simple » au niveau de sa complexité algorithmique. La règle dont on s'abstient dans un premier temps est par la suite appliquée à la solution du sous-problème pour trouver une solution optimale à notre problème d'origine. Ici on supprime la règle du poids maximal.

1 Algorithme de résolution exacte

1.1 Construction du tableau

Nous allons construire un tableau tel que : chaque ligne représente le nombre d'objets que l'on se donne. A savoir qu'à la nième ligne, on s'autorise uniquement à utiliser les (n-1) premiers objets de notre liste. Chaque colonne représente une valeur à atteindre (j-1). Ainsi la case située à la i-ième ligne et à la j-ième colonne représente le poids minimal nécéssaire pour atteindre la valeur (j-1) avec les (i-1) premiers objets. Le tableau est rempli en partant de la première ligne. Si une case est infinie, il n'y a alors aucune combinaison d'objets permettant d'atteindre la valeur (j-1). La première ligne est initialisée de la manière suivante (dans la première ligne on ne possède aucun objet) : La case (0, 0) est mise à

0 (première ligne première colonne) et les autres cases sont mises à l'infini. Ensuite, on remplit les lignes suivantes en utilisant la méthode ci-dessous.

La ligne i correspond à la ligne qui vient d'être complétée.
Pour v allant de 0 à Vmax faire :
V(i+1) est la valeur de l'objet que l'on ajoute à la liste d'objet pour cette ligne
P(i+1) est le poids de l'objet que l'on ajoute à la liste d'objet pour cette ligne
P(i, v) est le poids de la case sur laquelle on est
P(i+1, v) est le poids de la case en dessous de celle sur laquelle on est
$P(i+1, v) = minimum(P(i, v),\ P(i+1) + P(i, v - v(i+1)))$

1.2 Construction de la solution

Nous avons maintenant un tableau dont la dernière ligne nous donne pour la valeur v le poids minimal nécessaire pour l'atteindre.

Pour trouver la valeur de la solution respectant la contrainte de poids, on parcourt la ligne de droite à gauche. L'indice de la colonne correspondant au premier poids inférieur à la valeur maximale de notre sac à dos est la valeur maximale que l'ont peut atteindre. On note la valeur Vopt.

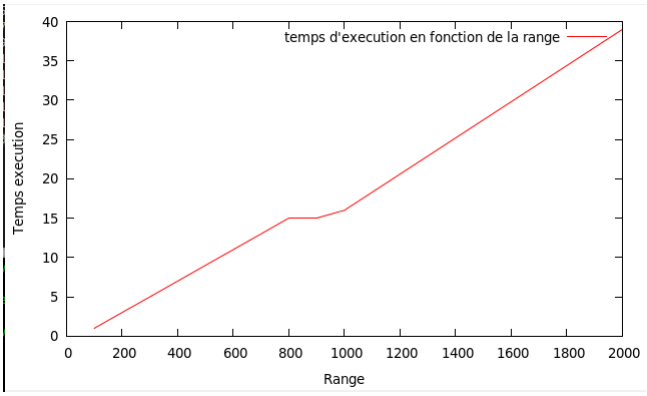
On doit maintenant reconstruire notre solution (retrouver les objets que l'on a mis dans notre sac à dos pour atteindre cette valeur). Pour ce faire, on remonte la colonne Vopt de bas en haut. A chaque fois que P(i, Vopt) est different de P(i-1, Vopt), on ajoute l'objet à la i-ième position dans notre liste. Arrivé en haut, on a récupéré l'ensemble des objets à mettre dans notre sac à dos pour atteindre la valeur maximale.

1.3 Analyse de la complexité et benchmark

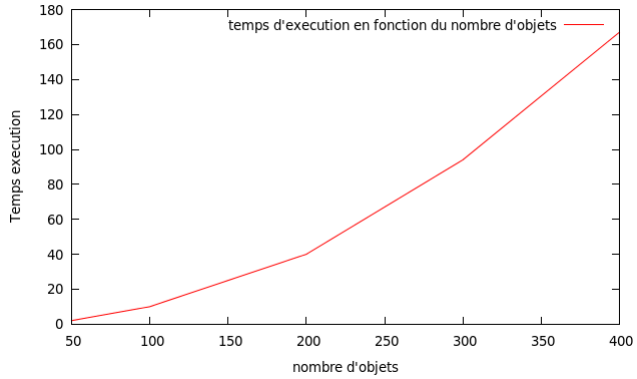
La taille du tableau est de n*Vmax ou n est le nombre d'objets + 1 et Vmax est la somme des valeurs des objets (car on ne pourra jamais dépasser une telle valeur dans un sac à dos). Pour chaque case on effectue 1 calcul + une comparaison. On a donc un résultat en temps constant. La complexité de cet algorithme est donc en O(n*Vmax), tout en sachant que la valeur de Vmax est dépendante de n.

Pour tester notre algorithme nous allons donc essayer d'influer sur deux paramètres, la valeur de Vmax en augmentant l'amplitude possible pour la valeur de nos objets, puis le nombre d'objets.

En traçant la courbe des variations du temps en fonction de l'amplitude des objets (en faisant varier uniquement Vmax) on obtient la courbe suivante qui semble varier de manière linéaire. Le nombre d'objets ayant été fixé à 20 pour les tests.



On trace ensuite la courbe du temps d'execution de l'algorithme pseudo-polynomial en fonction du nombre d'objet. L'amplitude de la valeur des objet étant fixée à 20. La courbe semble évoluer alors selon une puissance (facteur polynomial).



Le temps d'exécution pour 400 objets est de 167 secondes sur un netbook, soit 2 minutes et 47 secondes. on ne peut donc pas espérer atteindre 1000 objets dans un temps raisonnable.

2 Algorithme d'approximation fortement polynomial

2.1 Principe de fonctionnement

Pour cet algorithme, nous allons reprendre l'algorithme précédent. Mais on va appliquer un traitement aux objets avant de construire le tableau dans le but de réduire Vmax.

On va donc appliquer une normalisation pour passer d'objets de valeurs $V \in [a, b]$ à des objets de valeur $v \in [a', b']$ avec $|b' - a'| < |b - a|$. Ainsi on va réduire la taille du tableau. En contrepartie la solution sera moins précise. Par exemple si on a deux objets de valeur 1001 et 1002, ils pourront après

normalisation avoir la même valeur, ce qui pourrait donner des résultats non-optimaux.

La normalisation est faite de la manière suivante :

$$v' = \lfloor \frac{vi}{k} \rfloor$$

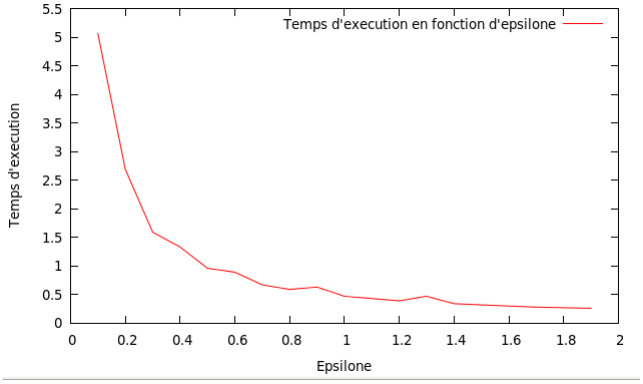
avec :

$$k = \frac{\varepsilon V_{max}}{n}$$

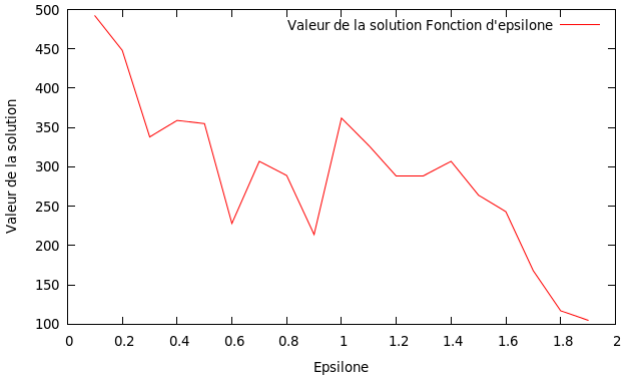
où ε va définir la précision de notre solution. La solution sera (1-ε) approchée

2.2 Benchmark

On fixe pour les test l'amplitude et le nombre d'objets. On teste deux critères : la précision de la solution et la rapidité d'exécution de l'algorithme. On trace pour commencer la courbe représentant l'évolution du temps d'éxécution de l'algorithme pseudo-polynomial approché en fonction de la valeur d'epsilone :



Puis on trace la courbe de la qualité de la solution en fonction de la valeur d'epsilone, la valeur optimale étant de 497 :



On retrouve donc bien le fait que plus epsilone augmente plus le temps d'exécution diminue et la qualité de la solution diminue.

Deuxième partie

Algorithmes arborescents

1 Formulation linéaire

Pour le problème du sac à dos qui nous intéresse, on propose le programme linéaire ci-contre :

Soit $x_j \in \{0,1\}$ pour $j \in \{0,n\}$ et $j \in \mathbb{N}$ tel que si $x_j = 0$ on ne prend pas l'objet j , sinon on le prend.

Soit U_j l'utilité (la valeur) de l'objet j.

Soit P_j le poids de l'objet j.

Soit b le poids maximal que l'on peut mettre dans le sac à dos.

Alors notre problème de maximisation est :

$$Max \sum U_j x_j$$

Et la contrainte est :

$$\sum P_j x_j \leq b$$

Si on relâche les contraintes du problème et que $x_j \in [0,1]$, alors, pour avoir une solution optimale, on prend les premiers éléments de la liste d'objets triés par densité de valeur, c'est à dire triés par rapport à leur $\frac{U_j}{P_j}$. Cette solution est alors supérieure ou égale à celles que l'on pourra trouver avec le problème « discret ». On a donc une borne supérieure facilement calculable. Si on repart de notre liste triée en fonction de leur densité et qu'on applique un algorithme glouton (on prend des objets dès qu'on peut les ajouter à notre sac), on obtient alors une borne minimale.

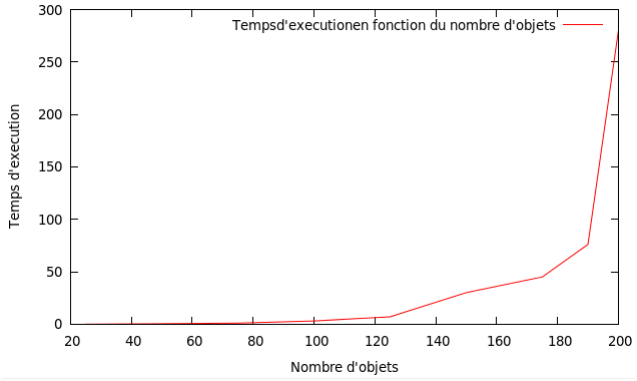
2 Algorithme arborescent exact

2.1 Principe de fonctionnement

L'algorithme arborescent exact agit de la manière suivante : pour chaque noeud on calcule une borne maximale et on recoit une borne minimale de son père. Si la borne minimale est inférieure à notre borne maximale alors on ne pourra pas trouver une meilleur solution. On réalise alors une coupure (on n'explore pas les fils). Sinon si notre borne minimale est supérieure à la borne minimale de notre père on retourne notre borne minimale. Dans le cas contraire, on retourne la borne minimale de notre père. A chaque noeud on va à droite si on prend l'objet et à gauche si on ne le prend pas. Les objets sont triés par densité pour permettre d'utiliser les bornes minimales et maximales trouvées dans la première partie.

2.2 Benchmark

Ici notre algorithme ne dépend pas de la valeur des objets. On réalise donc des tests faisant uniquement varier le nombre d'objets. Les courbes sont faites à partir de la moyenne de 5 tests pour chaque valeur. En fonction du tirage des objets le temps d'exécution est très différent du fait des coupures. En effet si par exemple la racine de l'arbre coupe directement la branche de droite on se retrouve avec seulement la moitié des possibilité à explorer.



3 Randomized Adaptative Search Procedure

3.1 Principe de fonctionnement

Dans cet algorithme, on parcourt aléatoirement l'arbre jusqu'à trouver une solution. Si cette solution (arriver à une feuille) est meilleure que la meilleur solution trouvée précédemment, on la garde. On répéte la procédure durant un temps T qui est le paramètre de « précision » de l'algorithme. Le parcourt de l'arbre est donc : si je ne peux pas prendre l'objet (car l'objet est trop lourd) je vais sur la branche impliquant que je n'ai pas pris l'objet. Sinon j'ai une chance sur deux de prendre l'objet (une chance sur deux d'aller à droite ou à gauche dans l'arbre).

3.2 Benchmark

Cet algorithme à l'avantage de pouvoir être paramétré selon son temps d'exécution. Par contre la valeur approchée au pire des cas est très mauvaise : on utilise de l'aléatoire. Nous avons 50% de chance de ne pas prendre l'objet.

Il existe donc une faible chance (très faible mais quand même existante) de ne jamais prendre aucun objet durant l'ensemble des parcourts de l'arbre. Dans un tel cas la valeur optimale retournée serait de 0. La valeur optimale étant de n, la solution serait alors approchée à $\frac{n}{0^+} = \infty$dans le pire des cas.

En réalité, il n'existe pas de générateur de valeur aléatoire. On utilise donc un générateur de valeurs pseudo-aléatoires à distribution linéaire. On ne peut

donc pas avoir un tel cas de figure. On trace la courbe représentant la valeur de la solution en fonction du temps de calcul (la solution optimale est de 308) :